Implementation of Lewis’s Airline

Functions

Query Flights

This function was made so that the aggregators could easily request all of the flights from the airline database given a date, arrival location, departure location, the number of passengers wanting to book, and given the option to supply a maximum price they are willing to pay per ticket. If a maximum price is not supplied then every flight at any price will be returned to the aggregator. All of the fields other than the maximum price are mandatory and must be sent by the aggregators to get a response from the API.

This function takes in a request as its only parameter, which is sent by entering the variables into the URL. Therefore, the function receives the query strings which then uses these to filter through the database to find if there are any matching flights with these parameters. There are two possible searches through the database depending if the aggregator supplied a max price as a query string. If there was a query string supplied, then the search through the database will contain this max price filter to ensure the flights returned are not over the specified limit.

The search containing all the query strings supplied by the aggregators is used to filter through the Flight Instance table of the database. This is the only table that holds information that is necessary for this query so only this table is filtered through or accessed. If the search on the table has no results then a 404 error code is returned to the aggregator as the requested resource doesn’t exist. Otherwise, if the filter does return at least one flight matching the query strings, the Flight Instances are then passed to the Flight Serializer to convert the model instance to a native Python datatype that can be worked with easily.

The query flights function only accepts GET requests and any other request sent will result in a 400 bad request error being thrown. This is because the API has detected that there has been an error from the client and the request can’t be processed.

The fields this function returns are all the fields in the Flight Instance table which are id, plane_id, flight_ticket_cost, departure_country, arrival_country, departure_time, arrival_time, num_available_seats, airline_name. Also, a status code of 200 to tell the aggregator that the query was successful.

Query Seats

The query seats function is used to get all the Seat Instances from a specific flight. This function only accepts GET requests and takes a request similar to the query flights function as the variables are sent in the URL as query strings. If the method is anything other than a request then an error is returned with the status code of 400 as it is a bad request. The request must contain the flight_id of the flight that the seats are being queried from. This is the only variable needed for this function as just the id can identify any flight in the database.

Similar to the query flights function, the filter function is used to find every Seat Instance with the flight_id sent from the aggregator, and all the seats with the same flight_id are returned, along with a status code of 200 to indicate the query was successful. Each Seat Instance that is returned contains the id, the seat_name, an available boolean, and the flight_id for the flight it is linked to. If the filter finds no seat, similarly to the flight query an error is returned with the status code of 404 as the requested resources don’t exist.

Update Seats

The update seats function is used to get a passenger with a seat they want to change and then change it to the new seat. To do this, a PUT request is required as the database is being updated. If any other method is used, a response is returned with the error code 400 indicating it is a bad request.

This function only takes one parameter, which is the request. But, unlike the other functions so far, is sent having a body of JSON rather than data being sent using query strings. The request must have the variables booking_id, seat_name for the new seat, and finally the first_name and last_name of the passenger that wants to change their allocated seat.

Using these variables, the Passenger object, the old Seat Instance, and the new Seat Instance to which the user wants to change their allocated seat to are requested from the database. The old Seat Instance has its availability boolean changed to True and the Seat Instance they want to change to has its availability changed to False. Then, the Passenger object has its seat_id updated to link to the Seat Instance the booking is changing to.

If all of this is done successfully, a response is sent to the aggregator with the status code 200 indicating the seat update was successful.

Add Booking

The add booking function is the first function that has been explained so far which sends requests to the payment service. This uses the requests library that must be imported into the Python code and installed onto the virtual environment on PythonAnywhere.

This function only takes POST requests as its only function is to create new bookings. If any other request method is used, similar to the previous functions, a response with an error code of 400 will be returned as it is deemed to be a bad request.

If the method of the request is a POST, the function then parses the data it has been sent by the aggregator in the request. Inside the request, there are a lot of different variables sent by the aggregator so they will be explained in other parts of this section.

The first three variables are flight_id, lead_passenger_contact_email, and lead_passenger_contact_number. The flight_id is used to find data from the flight such as the ticket cost so it can be used in the calculation of the total booking cost. It is also used later to filter for the Seat Instances. The other two variables are saved in the booking and are used when deleting the booking to validate that the correct user is deleting a booking.

Next, a list of dictionaries of passenger information for each individual passenger is used to store each one of their data as a separate Passenger object. The Passenger object includes a link to their Booking Instance using the booking_id, their first and last name, date of birth, nationality, passport number, and a seat_id which links to their Seat Instance for the seat they have booked. All of these variables are passed to the Passenger Serializer to change them to a Python data type, checked if they are valid, and then a new Passenger object is created for each passenger. Now a new passenger has been added to the flight other tables' data must be updated such as the availability of the Seat Instance that they have just booked and the number of available seats, which is a variable stored in each Flight Instance. So before moving on, both of these variables are updated and then the updated objects are saved.

Next in the request, is a dictionary of payment information that is being used to pay for the flight. The data is parsed from the request into a new dictionary which also contains the airline's banking information as this all needs to be together so it can be sent to the payment

APIs. The details that go into the card_details dictionary, which is the dictionary sent to the payment APIs, are the sender's cardholder name, the sender's card number that has been hashed, the sender's cvc which has also been hashed, the sender's sort code, the sender's expiry date of their card, the airline's cardholder name, the airline's sort code, the airline's account number, and the payment amount for the booking. Most of these details come either from the request sent from the aggregator, or the global variables storing the airline's banking information. Whereas, the payment amount must be calculated using the number of passengers in the booking and the cost per ticket by multiplying them together. This amount must be saved if the booking is deleted and the customer needs to be refunded, as if the ticket cost changes, then the new value calculated may be different from the original amount the customer paid.

Now that the dictionary is created, the request must be sent to the correct payment APIs which is done by checking the sender's sort code. Depending on which sort code decides which payment API to send the request to. After the payment is complete, the payment API responds with the transaction_id which is saved in the Booking Instance, and the payment_confirmed variable is set to True.

Finally, if all that has happened as planned a response with status code 200 is returned to the aggregators as the booking was successful.

Get Booking Details

Get booking details is the final GET request function that was created for this API and its function is to return the booking details of a booking after being given its booking_id. This function is used by the aggregators when in the manage booking section of their implementation.

Like the first two functions that were discussed in this documentation, this again is a GET function so only a request with the GET method is able to be used and any other will return a response with an error code of 400 as it is a bad request.

In the GET request sent from the aggregators, only the booking_id and the lead_passenger_contact_email are needed for the function to find and return the booking details of the booking. This is extra information than the minimum as the booking_id by itself is enough to find the booking but the lead_passenger_contact_email is there for validation purposes so only the actual customer can see their own booking information.

This function returns all the variables in the Booking Instance of booking_id which was sent by the aggregator which is sent to the Booking Serializer to turn the request from the aggregator into a Python dictionary. Also, now it is a Python dictionary, the number of passengers in the booking and the flight_id for the flight that the booking is for is also added after they have been collected from the other tables in the database.

To find the flight_id of the connected flight to the booking, the function must find a passenger in the booking in the Passenger table, then their seat from the Seat Instance table, then the connected flight in the Flight Instance table. Even though this seems a long route to find the flight_id, this is the fastest route from the bookings table to the flight instances table.

If the Booking Serializer is valid and the extra data is added into the dictionary correctly the data is returned to the aggregator, along with a status code of 200 confirming the GET request was a success.

Delete Booking

The delete booking function is the final function implemented for this API and its function is to delete a booking when a user no longer wants it and to correctly restore the previous data in the other associated tables, such as the Seat Instances being made available again and the number of passengers on a flight being increased. Then it finishes off by sending the money back to the customer for the same amount as they initially paid for their booking. In simple terms, it does the reverse of the add booking function.

Initially, the request method must be DELETE, which makes this function the only function that accepts the delete message in this API. If it is not a DELETE method then a response with the status code 400 is returned to the aggregator to indicate there has been an error.

Next, the function parses the data it received from the aggregator into variables, such as the booking_id and the lead_passenger_contact_email. Alike the add booking function, this is so they can be used to locate the booking that is being deleted and validate that it is the correct user deleting it. If the email wasn't checked before the booking was deleted any user could delete other bookings and steal the refund by having it sent to their own bank account.

Now the email has been validated, the Passengers linked to the Booking Instance that is being deleted are looped through and their Seat Instance is set back to available, and the number of seats available for the Flight Instance is incremented for each Passenger. After the data has been changed in both tables, both objects are saved.

Another variable in the request from the aggregator, the transaction_id from the original payment of the booking, is used in the request to get the booking details from the original booking. The booking details returned from the payment API are then used to create another Python dictionary that is sent to the payment API again to make the payment back to the customer. This new pay request should be the exact opposite request that was originally done when the booking was first paid for.

After the refund has been successfully paid to the customer, the payment API should return a status code of 200. Finally, the Booking Instance and the Passengers linked to it can be deleted and a response with a status code of 200 can be returned to the aggregator informing them of the successful deletion of the booking and a successful refund of the customer's money.

Populating the Database

As the process of populating a database is a slow process, a Python script was created to increase the speed and accuracy. In this section, the population of each table in the database will be explained and the thought process that was behind each decision.

Populate Countries

Populating the Countries table was a simple process as we had created a CSV file of all the countries in the world in alphabetical order, along with the continent the country is in and their longitudes and latitudes. So when populating the table, the script simply reads through the CSV file and creates a new Country object for each country.

Populate Planes

Similar to the Countries table, populating the Planes table was a simple task but was used to add much more depth to my implementation as I created multiple different sizes of planes.

Each plane had a maximum distance it could fly and a maximum capacity of seats it had. This will be used later in other tables to decide which plane will be chosen for each flight created.

Populate Flights

The population of the Flights table is the most complex function in the population script as it uses many of the other tables and randomises different options so when ran many times, many different flights are created.

The options that can be used to change the randomisation of the data include adding a date range between when the flights will be created, the range in which the ticket costs will be, and the number of new flights that will be created.

After these fields have been set the script can be run which first creates a new Flight Instance for the new flight it will generate. When generating a flight, there are many different variables that will be randomised. The first variables are the departure and arrival destinations of the flight which is done by generating a number between 1 and 196 for both of them and then getting the country from the Country table with that id. It is first ensured that they are not the same number as for this implementation, domestic flights are not available.

Next, the ticket cost of the flight is simply generated by obtaining a random integer between 50-500 using the random function in Python. Then, the decision of which plane will take the flight is selected. This is done by calculating the distance between the two countries using the longitude and latitude found in the Country table. The geopy.distance library is imported to do this calculation and depending on the result, a plane is chosen that has a maximum distance closest to the number found in the calculation. Finally, a random date is calculated in the date range between the two previously inputted dates and saved in the Flight Instance. All this is run for each flight and as it is inside a for loop this can be run as many times as needed to keep generating new flights.

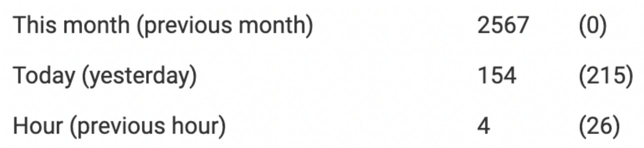
Populate Seats

The function to populate the seats simply goes through the Flight Instances one at a time that was created in the previous function. For each Flight Instance, the maximum number of seats for the plane is used to know how many Seat Instances are needed and then a loop is used to create the seats. Every plane has rows of size six but depending on the size of the plane, there will be a different number of rows.

Testing and validity

Testing the airline is crucial as it is the connecting part of the entire system between the aggregators and the payment systems. Therefore it was crucial for my implementation to be online of PythonAnywhere as soon as it could so that testing between all the systems could begin with plenty of time before the submission deadline.

To show the extensive testing that has been carried out between the different systems, a screenshot of the traffic to the PythonAnywhere will be shown to demonstrate how many requests have been made to just my airline before the URL was changed.



This month (previous month)	2567	(0)
Today (yesterday)	154	(215)
Hour (previous hour)	4	(26)